Programming a Distributed System Using Shared Objects

Tanenbaum, A.S.; Bal, H.E.; Kaashoek, M.F.

published in
Proc. 2nd Int'l Symposium on High-Performance Distributed Computing
1993

document version
Publisher's PDF, also known as Version of record

Programming a Distributed System Using Shared Objects

Andrew S. Tanenbaum
Henri E. Bal

Dept. of Mathematics and Computer Science, Vrije Universiteit
Amsterdam, The Netherlands

M. Frans Kaashoek

Laboratory for Computer Science, M.I.T.
Cambridge, MA

Email: ast@cs.vu.nl, bal@cs.vu.nl, kaashoek@lcs.mit.edu

Abstract

Building the hardware for a high-performance distributed computer system is a lot easier than building its software. In this paper we describe a model for programming distributed systems based on abstract data types that can be replicated on all machines that need them. Read operations are done locally, without requiring network traffic. Writes can be done using a reliable broadcast algorithm if the hardware supports broadcasting; otherwise, a point-to-point protocol is used. We have built such a system based on the Amoeba microkernel, and implemented a language, Orca, on top of it. For Orca applications that have a high ratio of reads to writes, we have measured good speedups on a system with 16 processors.

1. Introduction

As CPU prices continue to drop, more and more systems will be constructed from multiple CPUs. The techniques for building the hardware for such systems are beginning to be understood, but the software is in a more primitive state. In this paper we will describe our model for writing software for highly parallel computers, and discuss some example programs and measurements.

Two kinds of multiple CPU systems exist: multiprocessors and multicomputers. A *multiprocessor* is a machine with multiple CPUs that share a single common virtual address space. All CPUs can read and write every location in this address space. Multiprocessors can be programmed using well-established techniques, but they are difficult and expensive to build. For this reason, many multiple CPU systems are simply a collection of independent CPU-memory pairs, connected by a communication network. Machines of this type that do not share primary memory are called *multicomputers*. Because these machines are easier to build and are likely to dominate the highly parallel computer market in the future, our model has been designed for this class of machines, but our software also works on multiprocessors.

The usual approach to programming a multicomputer is message passing. The operating system provides primitives SEND and RECEIVE in one form or another, and programmers can use these for interprocess communication. This makes I/O the central paradigm for multicomputer software, something that is unfamiliar and unnatural for many programmers.

An alternative approach is to simulate shared memory on multicomputers. One of the pioneering efforts in this direction was the work of Li and Hudak [12]. In their system, Ivy, a collection of workstations on a local area network shared a single, paged, virtual address space. The pages are distributed among the workstations. When a CPU references a page that is not present locally, it gets a page fault. The page fault handler then determines which CPU has the needed page and sends it a request. The CPU replies by sending the page. Although various optimizations are possible, the performance of these systems is often inadequate.

Another approach is not to split the shared address space up into fixed-size pages, but into programmer-defined objects. These objects are then sharable among multiple machines. Linda [5], for example, is based on an abstract tuple space. A process can place a tuple in the tuple space, and another process on a different processor can remove it. In this way processes can communicate with one another.

In Emerald [8] location-independent abstract data types can be defined. Processes can perform operations on an abstract data type no matter on which machine it is located. It is up to the system to send the operation to the data or the data to the requesting machine.

Both tuple-based and abstract data type-based schemes eliminate the problem found in Ivy and similar systems of having to move fixed-size units (e.g. 8K pages) around, but they they have other problems. Emerald does not replicate data, which can lead to performance problems; Linda has fixed primitives that are low-level and inflexible.

This work was supported in part by the Netherlands Organization for Scientific Research as part of its *Pionier* program.

2. Shared Data-Objects

Our design is based on the idea of doing parallel programming on distributed systems using shared data-objects. These objects may be replicated on multiple processors, and are kept synchronized by system software, the runtime system, as shown in Fig. 1 Associated with each shared object is a set of operations that are encapsulated with the object to form an abstract data type. Objects are managed by a runtime system, as shown in Fig. 1.

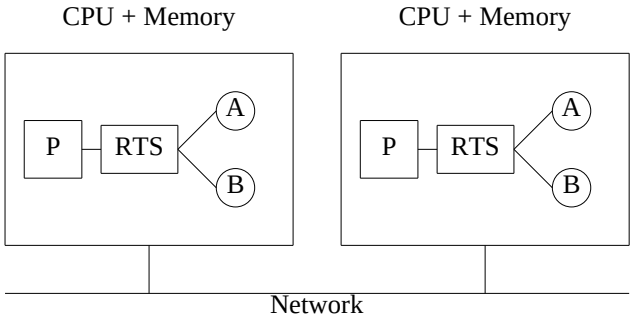

Fig. 1 An object can be replicated on each machine. P = process, RTS = runtime system, A and B are shared data-objects. To the user processes, A and B look like they are in physical shared memory.

Processes on different machines can perform operations on shared objects as though they were in physical shared memory. Shared objects exhibit the property of sequential consistency in that if processes simultaneously perform operations on a shared object, the final result will look like all the operations were performed in some sequential order [11]. The order is chosen nondeterministically, but all processes will see the same order of events and the same final result. It is up to the runtime system to maintain this illusion.

By encapsulating the data inside abstract data types, we insure that processes may not access shared data without the runtime system gaining control. Getting control is essential to make sure objects are consistent when accessed and to guarantee that updates are propagated to other machines in a consistent manner. These properties are not available with a page-based distributed shared memory in which any process can touch any virtual address at will.

Replicating shared objects has two advantages over systems like Ivy. First, reads to any object can be done locally on any machine having a replica. No network traffic is generated. (For our purposes, a read is an operation that does not change the state of its object.) Second, more parallelism is possible on reads, since multiple machines can be reading an object at the same time, even if the object is writable. With page-based schemes, having many copies of writable pages is possible only under restricted circumstances involving weakened consistency.

Whether replication can be done efficiently in software depends on two factors. The first is the ratio of reads to writes. If the vast majority of accesses to shared data are reads, then having a copy of each shared object on each machine that needs it is a good idea. The gain from making reads cheap generally results in a major gain in performance. The other factor is how expensive writes are. If writes are exceedingly expensive (in terms of delay, bandwidth, or computing power required), even a moderately high ratio of reads to writes may not be enough to make replication worthwhile. We have studied this question in detail and reported on it elsewhere [3].

3. Implementation

The idea of sharing objects on a distributed system stands or falls with the efficiency of its implementation. If it can be implemented efficiently, high performance parallel systems can be built as multicomputers and programmed as multiprocessors, combining the hardware simplicity of the former with the software simplicity of the latter. If it cannot be implemented efficiently, the idea is of little practical value. Our results show that shared objects can be implemented efficiently under certain circumstances, and gives good results for a variety of problems.

The system described in this paper consists of three major components: The Amoeba microkernel. The shared object runtime system. The Orca parallel programming language. We will discuss each of these in turn in this section.

3.1. The Amoeba microkernel

Amoeba is a distributed operating system consisting of a microkernel and a collection of server processes [15]. Amoeba was designed for a large number of machines, called the processor pool, called by a (broadcast) network. There are also machines for handling specialized servers, such as the file system, as well as workstations for clients, but the real computing is done on the processor pool machines. A copy of the microkernel runs on all these machines.

The Amoeba microkernel has four primary functions:

1. Manage processes and threads.
2. Provide low-level memory management support.
3. Handle I/O.
4. Support transparent communication.

Let us consider each of these in turn.

Like most operating systems, Amoeba supports the concept of a process. In addition, Amoeba also supports multiple threads of control, or just *threads* for short, within a single address space. A process with one thread is essentially the same as a process in UNIX®. Such a process has a single address space, a set of registers, a program counter, and a stack.

In parallel applications, processes often have multiple threads. Threads can synchronize using semaphores and

mutexes to prevent two threads from accessing critical regions or data simultaneously.

The second task of the microkernel is to provide low-level memory management. Threads can allocate and deallocate blocks of memory, called *segments*. These segments can be read and written, and can be mapped into and out of the address space of the process to which the calling thread belongs. To provide maximum communication performance, all segments are memory resident.

The third job of the microkernel is to provide the ability for one thread to communicate transparently with another thread, regardless of the nature or location of the two threads. The model used here is remote procedure call (RPC) between a client and a server [4].

All RPCs are from one thread to another. User-to-user, user-to-kernel, and kernel-to-kernel communication all occur. When a thread blocks awaiting the reply, other threads in the same process that are not logically blocked may be scheduled and run.

The third basic function of the microkernel is to manage I/O devices, handle interrupts, and so on. Device drivers run as threads within the kernel. All other functionality is located in user-space servers and other processes.

Totally-ordered broadcasting

Amoeba also provides totally-ordered, reliable broadcasting on unreliable networks through use of a software protocol [9]. The protocol supports reliable broadcasting, in the sense that in the absence of processor crashes, the protocol guarantees that all broadcast messages will be delivered, and all machines will see all broadcasts in exactly the same order, a property useful for guaranteeing sequential consisting. This feature is heavily used by higher layers of software.

When an application starts up on Amoeba, one of the machines (which normally have identical hardware) is elected as *sequencer* (like a committee electing a chairman). If the sequencer machine subsequently crashes, the remaining members elect a new one. We have devised and implemented two different reliable broadcast algorithms having slightly different properties. In the first algorithm, called *PB* (Point to point - Broadcast), a runtime system needing to reliably broadcast a message (e.g., a new value of an object) traps to the kernel. The kernel adds a protocol header containing a unique identifier, the number of the last broadcast it has received, and a field saying that this is a *RequestForBroadcast* message and sends it as a point-to-point message to the sequencer. When the sequencer gets the message it adds the lowest unassigned sequence number, stores the message in its history buffer, and then broadcasts it.

When such a broadcast arrives at each of the other machines, a check is made to see if this is the next message in sequence. If this is message 25 and the previous message received was 23, for example, the message is temporarily buffered and a request is sent to the sequencer asking it for message 24 (stored in the sequencer's history buffer). When 24 comes in, 24 and 25 are passed to the application program in that order. Under no circumstances are messages passed to application programs out of order. This is the basic mechanism by which it is guaranteed that all broadcasts are seen by all machines, and in the same order.

The other reliable broadcast algorithm is called *BB* (Broadcast - Broadcast). In this method, the user broadcasts the message, including a unique identifier. When the sequencer sees this, it broadcasts a special *Accept* message containing the unique identifier and its newly assigned sequence number. A broadcast is only official when the *Accept* message has been sent.

These protocols are logically equivalent, but they have different performance characteristics. In *PB*, each message appears in full on the network twice: once to the sequencer and once from the sequencer. Thus a message of length *m* bytes consumes 2*m* bytes worth of network bandwidth. However, only the second of these is broadcast, so each user machine is only interrupted once (for the second message).

In *BB*, the full message only appears once on the network, plus a very short *Accept* message from the sequencer, so only half the bandwidth is consumed. On the other hand, every machine is interrupted twice, once for the message and once for the *Accept*. Thus *PB* wastes bandwidth to reduce interrupts compared to *BB*. The present implementation looks at each message and depending on the amount of data to be sent, dynamically chooses either *PB* or *BB*, using the former for short messages and the latter for long ones (over 1 packet).

3.2. The shared object runtime system

We have developed two runtime systems to use with Amoeba and Orca. The first one is used on networks that have hardware broadcasting or multicasting. The second one is used on systems that have neither of these. Both are described below.

3.2.1. Reliable broadcasting

When the underlying network provides unreliable broadcasting or multicasting, the *PB* and *BB* protocols provided by Amoeba can be used to make broadcasting reliable and totally ordered. The runtime system in this case is straightforward. In the initial implementation, every object is replicated on all machines that need it (an optimizing scheme using partial replication is under development). Reads are then done locally. Writes are done either by doing the operation on the machine doing the write and then broadcasting the result, or by broadcasting the operation code and parameters and letting each machine run the operation itself.

The broadcast RTS has an object manager for each machine. Reads are done directly, bypassing the object manager. Writes are done by asking the local object

manager to do the work. Incoming broadcasts are handled by the object managers, which process them in strict FIFO (i.e., sequence number) order, thus guaranteeing that writes are seen by all processes in the system in the same order (necessary for enforcing sequential consistency). When a broadcast packet comes in, the local object is locked to prevent local reads during the update.

3.2.2. The point-to-point runtime system

When the network does not support broadcasting, objects can still be replicated, but they have to be managed using point-to-point messages and completely different protocols. We have implemented and tested two kinds of point-to-point protocols: invalidation and update (the broadcast system always uses update, because it is cheap).

Both protocols allow multiple copies of all objects to exist, so reads can be done locally when a copy is available locally. When an invalidation protocol is being used and an object is written, all copies except one are invalidated (i.e., discarded). When an update protocol is being used and an object is written, all copies are updated to the new value. In general, updating an object is more expensive than invalidating it, since a larger message and more complicated protocol is needed. Also, if you do multiple consecutive writes, invalidation may require one message whereas update may require one per write. On the other hand, if an object is needed on a machine whose copy is no longer valid, a new copy must be fetched, which also has a cost associated with it.

Both our runtime system using invalidation and our runtime system using update are based on the concept of having a primary copy of each object from which all other copies (secondaries) are derived. In the invalidation scheme, when a write is done, a message is sent to the machine holding the primary copy. The runtime system on this machine then locks the object and sends invalidation messages to all the other copies. Each of these sends back an acknowledgement when its copy has been invalidated. When all the acknowledgements have been collected at the primary site, the object is unlocked and made available for reading and copying.

The runtime system using updates also has a primary copy, but the protocol is more complicated due to the need to make multiple simultaneous updates sequentially consistent. A two-phase protocol is used. When a write is done, a message is sent to the primary copy, whose runtime system locks the object and sends an update message to all other copies. This message contains the operation code and parameters to allow the other machines to perform the updates. This approach requires less bandwidth than updating the object at the primary site and sending the result to the other sites.

When each of the secondaries receives the update message, it locks the object, performs the operation, and sends back an acknowledgement, keeping the object still locked. When all the acknowledgements have arrived back at the primary, the second phase of the protocol is entered in which a message is sent to each copy saying that the object can be unlocked and used again for local reads. Reads that are attempted while an object is locked are suspended until it is unlocked.

The decision of where to replicate each object is done dynamically based on runtime statistics. Initially, only one copy of each object is maintained. As accesses to objects are made, statistics are maintained. When the ratio of reads to writes on any machine exceeds a certain threshold, the runtime system concludes that there are so many reads being done and so few writes that having a local copy is worthwhile. A message is sent to the primary to fetch a copy. Similarly, when this ratio falls below another threshold, the runtime system concludes that having a copy is not worth the trouble due to the large percentage of write operations. The local copy is then discarded.

Comparisons of update and invalidation did not show a clear winner. Which one is better depends on the problem being solved. Our experience suggests that updating is better more often than invalidation, but more research is needed.

3.3. Orca

While it is possible to program directly with shared objects, it is much more convenient to have language support for them [1]. For this reason, we have designed the Orca parallel programming language and written a compiler for it. Orca is a procedural language whose sequential statements are roughly similar to languages like C or Modula 2 but which also supports parallel processes and shared objects.

There are four guiding principles behind the Orca design:

- Transparency
- Semantic simplicity
- Sequential consistency
- Efficiency

By *transparency* we mean that programs (and programmers) should not be aware of where objects reside. Location management should be fully automatic. Furthermore, the programmer should not be aware of whether the program is running on a machine with physical shared memory or one with disjoint memories. The same program should run on both, unlike nearly all other languages for parallel programming, which are aimed at either one or the other, but not both. (Of course one can always simulate message passing on a multiprocessor, but this is often less than optimal, especially if there is heavy contention for locks and certain other locations.)

Semantic simplicity means that programmers should be able to form a simple mental model of how the shared memory works. Incoherent memory, in which reads to shared data sometimes return good values and sometimes stale (incorrect) ones, is ruled out by this principle.

Sequential consistency is an issue because in a parallel system, many events happen simultaneously. By making operations sequentially consistent, we guarantee that operations on objects are indivisible (i.e., atomic), and that the observed behavior is the same as some sequential execution would have been. Operations on objects are guaranteed not to be interleaved, which contributes to semantic simplicity, as does the fact that all machines are guaranteed to see exactly the same sequence of serial events. Thus the programmer's model is that the system supports operations. These may be invoked at any moment, but if any invocation would conflict with an operation currently taking place, the second operation will not begin until the first one has completed.

Finally, *efficiency* is also important, since we are proposing a system that can actually be used for solving real problems.

Now let us look at the principal aspects of Orca that relate to parallelism and shared objects. Parallelism is based on two orthogonal concepts: *processes* and *objects*. Processes are active entities that execute programs. They can be created and destroyed dynamically. It is possible to read in an integer, *n*, then execute a loop *n* times, creating a new process on each iteration. Thus the number of processes is not fixed at compile time, but is determined during execution.

The Orca construct for creating a new process is the

fork func(param, ...)

statement, which creates a new process running the procedure *func* with the specified parameters. The user may specify which processor to use, or use the standard default case of running it on the current processor. Objects may be passed as parameters (call by reference). A process may fork many times, passing the same objects to each of the children. This is how objects come to be shared among a collection of processes. There are no global objects in Orca.

Objects are passive. They do not contain processes or other active elements. Each object contains some data structures, along with definitions of one or more operations that use the data structures. The operations are defined by Orca procedures written by the programmer. An object has a specification part and an implementation part, similar in this respect to Ada® packages or Modula-2 modules. Orca is what is technically called *object based* (in contrast with object oriented) in that it supports encapsulated abstract data types, but without inheritance.

A common way of programming in Orca is the Replicated Worker Paradigm [5]. In this model, the main program starts out by creating a large number of identical worker processes, each getting the same objects as parameters, so they are shared among all the workers. Once the initialization phase is completed, the system consists of the main process, along with some number of identical worker processes, all of which share some objects. Processes can perform operations on any of their objects whenever they want to, without having to worry about all the mechanics of how many copies are stored and where, how updates take place, and so on. As far as the programmer is concerned, all the objects are effectively located in one big shared memory somewhere, but are protected by a kind of monitor that prevents multiple updates to an object at the same time.

4. Applications

In this section we will discuss our experiences in implementing various applications in Orca. For each application, we will describe the parallel algorithm and the shared objects used by the Orca program. In this way, we hope to give some insight in the usefulness of shared objects.

In addition, we will briefly consider performance issues of the applications and give some experimental performance results. The performance measurements were carried out on the Amoeba-based Orca system described in the previous section using the broadcast runtime system. The hardware we use is a collection of MC68030s connected by a 10 Mb/sec Ethernet.

The applications we will look at are: the Traveling Salesman Problem, the Arc Consistency Problem, computer chess, and Automatic Test Pattern Generation. Additional Orca applications are described in [2].

4.1. The traveling salesman problem

The Traveling Salesman Problem (TSP) is our favorite example for Orca, since it greatly benefits from object replication. The problem is to find the shortest route for a salesman that visits each of a number of cities exactly once.

We solve the problem using a parallel branch-and-bound algorithm, which is implemented in a replicated worker style program. The problem is split up into a large number of small jobs, each containing a partial (initial) route for the salesman. The jobs are generated by a *manager* process and are stored in a *JobQueue* object. Each available processor runs a *worker* process, which repeatedly takes a job from the queue and executes it. For TSP, executing a job involves searching all possible routes starting with the partial route stored in the job description.

The parallel program keeps track of the best solution found so far by any worker process. This value is used as a bound. A partial route that is already longer than the current best route cannot lead to a better solution, so its search can be abandoned. Significant portions of the search space can therefore be pruned.

The bound must be accessible to all workers, so it is stored in a shared object. This object is read very frequently and is written only when a new better route has been found. In practice, the object may be read millions of times and written only a few times.

In summary, the TSP program uses two important

objects: a global bound and a job queue. The global bound is replicated on all worker processors. Since it has a very high read/write ratio, most operations (i.e., the reads) are done locally, without requiring any communication. Write operations, which occur far less often, are broadcast. If one worker discovers a better value for the bound, all other workers are informed instantly, and can immediately use the new value for pruning parts of their search space. The indivisible operation that updates the object first checks if the new value actually is less than the current value, to prevent race conditions.

The job queue mainly has write operations, since both adding and deleting jobs changes the queue's internal data. The RTS described in this paper (the original one), replicates it on all machines, although keeping a single copy would be better. Despite the global replication of both objects, the speedup is still excellent, as can be seen from the figure.

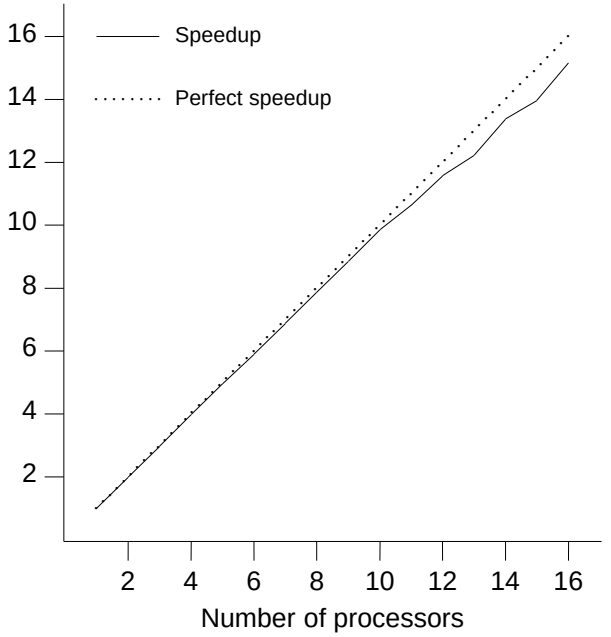

Fig. 2 Speedup for the Traveling Salesman Problem using a 14-city problem.

4.2. The arc consistency problem

The Arc Consistency Problem (ACP) is an important AI problem [14]. The input to the problem is a set of *variables* V_i, each of which can take a value from a domain D_i, and a list of *constraints*. Each constraint involves two variables and puts a restriction on the values these variables can have, for example $A < B$. The goal of ACP is to determine the maximal set of values each variable can take, such that all constraints are satisfied.

A straightforward sequential algorithm for solving ACP is as follows. Assign a set of possible values to each variable V_i; initially, the set for V_i contains all values in its domain D_i. Next, repeatedly restrict the sets using the constraints, until no more changes can be made. An obvious improvement is to keep a list of variables whose sets have been changed, and then only recheck constraints involving such variables.

For example, assume the current set for *A* is {1,10,100} and the set for *B* is {2,3,20}. The constraint $A < B$ can be used to delete the value 100 from *A*'s set. Now, all other constraints involving *A* have to be checked again.

A parallel implementation of the above algorithm is described by Conrad [6], using a message passing program that runs on an iPSC/2 hypercube. The parallel algorithm statically partitions the variables among the available processors. Each processor takes care of determining the value sets for the variables assigned to it.

We have implemented a similar program in Orca, but now using shared objects, and running on the Ethernet-based Amoeba system. The Orca program uses several shared objects. The sets associated with the variables are stored in a shared object, called *domain*. This object thus contains an array of sets, one for each variable. Operations exist for initializing the object, deleting an element from one of the sets, and set membership tests. The object is shared among all processes, since they all need to have this information.

Another object, called *work*, is used to keep track of which variables have to be rechecked. This object contains an array of Booleans, one per variable. If the value set of a variable *A* has been checked the entry for *A* is set to *false*. In addition, for all other variables *X* for which a constraint exists involving both *A* and *X*, the corresponding entry is also set to *true* if *A* was reduced.

The most complicated issue in parallel ACP is how to terminate the algorithm correctly. The algorithm should terminate if none of the variables need to be rechecked. Since the variables are distributed among the processors, however, testing this condition requires careful synchronization.

For this purpose, we use two shared objects. One object contains a Boolean variable that is set to *true* if a process discovers that no solution to the input problem exists, because one of its variables now has an empty set of values. Each process reads the object before doing new work, and quits if the value is *true*.

A second and more complicated object, called *result*, contains an array of Booleans, one per process. A process sets its entry to *true* if it is willing to terminate because it has no more work to do. The program can terminate if two conditions are satisfied: (1) all entries in the *work* object are *false*, and (2) all entries in the *result* object are *true*. In this case, no more work exists and neither will any process generate such work, so the program can safely terminate. The *work* and *result* objects have indivisible operations for testing these two conditions.

The speedups for ACP are shown in Fig. 3. The

program uses at least two processors, since the master process that distributes the work runs on a separate processor.

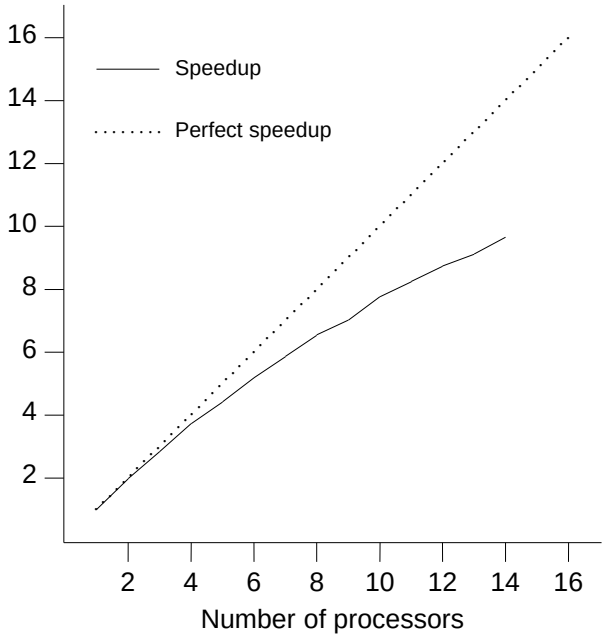

Fig. 3 Speedup for the Arc Consistency Problem using an input problem with 64 variables.

Although the program obtains significant speedups, the speedups are less than those reported for the original hypercube program. The objects used in the program are replicated on all processors, so there is a lot of CPU overhead in handling incoming update messages for these objects. We should also point out, however, that our program uses many operations to handle termination correctly, as explained above. The hypercube program uses a cheaper but far less elegant method to handle termination, based on time-outs.

4.3. Computer chess

Oracol is a chess problem solver written in Orca. It can be asked to look for "mate-in-N-moves" or for tactical combinations that win material. It does not consider positional characteristics.

Oracol's search algorithm is based on alpha-beta with iterative deepening and the quiescence search heuristic. Parallelism is obtained by dynamically partitioning the search tree among the different processors, using simple run-time heuristics. The program uses several shared objects (e.g., a job queue). We will only discuss two objects here, which are of particular interest.

The two objects implement a *killer table* and a *transposition table*. The killer table contains moves that turn out to cause many cutoffs in the alpha-beta algorithm. Killer moves are always considered first. The idea is that if, say, White threatens to capture a rook by playing "Queen to a8", then Black needs to do something against this threat. Any move by Black that does not prevent White from capturing the rook can immediately be refuted by the "Qa8" move, thus saving the trouble of much further analysis.

A transposition table is a table containing positions that have already been analyzed earlier during the search. The same board position can be encountered multiple times during the search, because different sequences of moves can result in the same position. The transposition table thus remembers positions and their evaluation values. Before a position is analyzed, the program first looks in its transposition table (using a hashing function), to determine if it has seen the position before.

Both the killer table and the transposition table can be implemented as local data structures or as shared objects. If used locally, no communication is needed, but processes cannot benefit from each other's tables. For example, a process may evaluate a position that has been evaluated before by another process. If the tables are shared, this will generally not happen, but now communication overhead is introduced for managing the shared tables.

In Orca, it is particularly easy to implement both versions and see which one is best. The tables are implemented using abstract data types (objects types). In the local version, each process declares its own instance of this type. In the shared version, only the main process declares a table object and passes it as a shared parameter to all other processes. The two versions differ in only a few lines of code. For Oracol, we have determined that, especially for the killer table, shared tables are most efficient.

The speedups obtained for the program are not very high, because alpha-beta is hard to parallelize efficiently. On 10 CPUs, we have measured speedups between 4.5 and 5.5. Almost all of the overhead is search overhead, which means that the parallel program searches far more nodes than a sequential program does.

4.4. Automatic test pattern generation

The largest program implemented in Orca so far (nearly 4000 lines) is for Automatic Test Pattern Generation (ATPG). ATPG is an important problem from electrical engineering. It generates test patterns for combinatorial circuits. Such a circuit consists of several input and output lines, and several internal gates (e.g., AND gates, OR gates). The output of a given circuit is completely determined by the input.

To test if a specific gate works correctly, the input lines must be set to certain values, such that the correct functioning of the gate can be determined from the output of the circuit. In other words, at least one of the output lines must be different for a correct and an incorrect gate. The problem is how to determine which inputs to use. In general, all gates of the circuit must be tested, so the problem becomes that of generating a set of input patterns

that together test the whole circuit. This problem is called the ATPG problem. The problem is NP complete, so in practice an ATPG program tries to cover as many gates as possible within the time limit imposed on it.

Many ATPG algorithms exist, and several parallel algorithms have been designed and implemented [10]. The Orca ATPG program is based on the PODEM algorithm [7]. The algorithm considers each gate in turn. It assigns values to certain input lines (determined by heuristic rules), and propagates these values through the circuit. If it discovers that the current assignment cannot lead to a test of the gate, it backtracks and tries alternative assignments.

The Orca program parallelizes ATPG by statically partitioning the fault set among the processors. Each processor is given a fixed number of gates, for which it computes the test patterns. Using this basic algorithm, the program achieves good speedups (close to linear) on circuits of reasonably large size.

An important optimization that can be applied to both the sequential and parallel ATPG algorithms is *fault simulation*. If a test pattern has been computed for a certain gate, this pattern will probably test other gates in the circuit as well. Fault simulation determines such gates and removes them from the list of gates for which patterns still have to be computed.

This optimization was easy to add to the Orca program. All processes share an object containing the gates for which test patterns have been generated. If a process adds an element to this set, all other processes also use it to determine which gates they can delete from their set. The Orca program using this optimization is faster in absolute speed (by about a factor of 3), but it obtains inferior speedups. This is partly due to the communication overhead, and partly to the fact that the static partitioning of work may now lead to a load balancing problem. We intend to use a more dynamic work distribution strategy in the future.

5. Summary

Shared objects offer the possibility of programming certain parallel applications on systems lacking physical shared memory. They offer a model comparable to what programmers of multiprocessors get to see. We believe that the shared object model allows systems to be built that have the ease of construction of multicomputers, combined with the ease of use of multiprocessors. For this reason, we see this model as a promising area for future research.

Acknowledgements

The Orca programs for ACP, computer chess, and ATPG, have been written by Irina Athanasiu, Robert-Jan Elias, and Klaas Brink, respectively. Jack Jansen wrote the point-to-point runtime system.

References

[1] Bal, H.E.: *Programming Distributed Systems*, Prentice Hall Int'l, Hemel Hempstead, UK, 1991.

[2] Bal, H.E., Kaashoek, M.F., and Tanenbaum, A.S.: ''Orca: A language for Parallel Programming of Distributed Systems,'' *IEEE Transaction on Software Engineering* vol. 18, pp. 190-205, March 1992.

[3] Bal, H.E., Kaashoek, M.F., Tanenbaum, A.S., and Jansen, J.: ''Replication Techniques for Speeding up Parallel Appl. on Distr. Systems,'' *Concurrency Prac. & Exp.*, vol. 4, pp. 337-355, Aug. 1992.

[4] Birrell, A.D. and Nelson, B.J.: ''Implementing Remote Procedure Calls,'' *ACM Trans. Computer Systems*, vol. 2, pp. 39-59, Feb. 1984.

[5] Carriero, N. and Gelernter, D.: ''Linda in Context,'' *Commun. ACM*, vol 32, pp. 444-458, April 1989.

[6] Conrad, J.M, and Agrawal, D.P.: ''A Graph Partitioning-Based Load Balancing Strategy for a Distributed Memory Machine,'' *Proc. 1992 Int. Conf. Parallel Processing (Vol. II)*, pp. 74-81, 1992.

[7] Goel, P.: ''An Implicit Enumeration Algorithm to Generate Tests for Combinational IC Circuits,'' *IEEE Trans. Computers*, vol. C-30, pp. 215-222, March 1981.

[8] Jul, E., Levy, H., Hutchinson, N., and Black, A.: ''Fine-Grained Mobility in the Emerald System,'' *ACM Trans. Computer Syst.*, vol 6, pp. 109-133, Feb. 1988.

[9] Kaashoek, M.F. ''Group Communication in Distributed Computer Systems,'' Ph.D Thesis, Vrije Universiteit, Amsterdam, 1992.

[10] Klenke, R.H., Williams, R.D., and Aylor, J.H.: ''Parallel-Processing Techniques for Automatic Test Pattern Generation,'' *IEEE Computer*, vol. 25, pp. 71-84, Jan. 1992.

[11] Lamport, L. ''How to Make a Multiprocessor that Correctly Executes Multiprocess Programs,'' *IEEE Trans. on Comp.*, vol. C-28, pp. 690-691, Sept. 1979.

[12] Li, K. and Hudak, P.: ''Memory Coherence in Shared Virtual Memory Systems,'' *ACM Trans. Computer Systems*, vol 7., pp. 321-359, Nov. 1989.

[13] Marsland, T.A., and Campbell, M.: ''Parallel Search of Strongly Ordered Game Trees,'' *Computing Surveys*, vol. 14, pp. 533-551, December 1982.

[14] Mackworth, A.K.: ''Consistency in Networks of Relations,'' *Artificial Intelligence*, vol. 8, pp. 99-118, Feb. 1977.

[15] Tanenbaum, A.S. et al., ''Experiences with the Amoeba Distributed Operating System,'' *Commun. of the ACM* vol. 33, pp. 46-63, Dec. 1990.